Recomputing normals for displacement and bump mapping, procedural style

Stefan Gustavson

November 26, 2021

Old style (hairy)

Bump mapping as originally proposed by Jim Blinn [1] was defined in 2-D parameter space (u, v) for parametric surfaces $\mathbf{P}(u, v)$ and a virtual displacement function $h(u, v)$. In order to use it with non-parametric surfaces, such as the polygon mesh models that are ubiquitous in modern computer graphics, some extrinsic parametrization is required. Fortunately, most modern applications of computer graphics use texture mapping, and the 2-D texture space (s, t) is such an extrinsic parametrization. The displacement can be specified as a single channel (gray scale) 2-D texture image $h(s, t)$, and the virtual displaced position $\mathbf{P}$ of the surface is a distance of $h(s, t)$ from the original position $\mathbf{P}_0$ along the normal direction $\hat{\mathbf{N}}_0$:

$$\mathbf{P} = \mathbf{P}_0 + h(s, t)\hat{\mathbf{N}}_0 \tag{1}$$

Recomputing the direction of the normal for the bumped surface requires computing of partial derivatives of $h(s, t)$ with respect to s and t: $\frac{\partial h}{\partial s}$ and $\frac{\partial h}{\partial t}$. For performance reasons, these derivatives may be precomputed and stored as a texture.

Normal mapping, a modern sibling to bump mapping, is performed similarly, only with the normal specified explicitly as a 3-component texture $\hat{\mathbf{N}}(s, t)$ which completely replaces the interpolated mesh normal. This is somewhat less flexible than bump mapping, but it requires less computations.

Displacement mapping, which involves actually changing the position of the surface, is closely related to bump mapping, because the displacement is mostly specified as taking place along the normal direction, as in Equation 1. Recomputing the normal after displacement mapping involves the same computations as bump mapping, requiring in-plane partial derivatives of the displacement function.

When applied to a plane, the corresponding remapping of the normal is quite straightforward for both bump mapping and normal mapping. However, when bump mapping is applied to a general surface with a locally varying normal direction, recomputing the normal requires full knowledge of the orientation of the surface in 3-D space. In addition to the ubiquitous normal direction, one or two additional vectors are specified to relate the 2-D texture coordinates (s,t) to 3-D space (x,y,z). The basis vectors $\hat{\mathbf{s}}$ and $\hat{\mathbf{t}}$ that define the tangent plane are orthogonal to the surface normal $\hat{\mathbf{N}}$ and parallel to the local gradient of each of the texture coordinates, ∇s and ∇t:

$$\mathbf{s} = \nabla s = (\frac{\partial s}{\partial x}, \frac{\partial s}{\partial y}, \frac{\partial s}{\partial z})$$

$$\mathbf{t} = \nabla t = (\frac{\partial t}{\partial x}, \frac{\partial t}{\partial y}, \frac{\partial t}{\partial z})$$

Unit vectors $\hat{\mathbf{s}}$ and $\hat{\mathbf{t}}$ are then computed by normalization of $\mathbf{s}$ and $\mathbf{t}$.

It should be noted that the computation of these vectors is usually not performed in this manner (using partial derivatives). Instead, the tangent plane vectors are specified explicitly in the form of additional vertex attributes for the mesh model. If the 2-D texture mapping has reasonably orthogonal coordinate directions, it may be enough to specify one of the vectors and compute the other by a cross product with the normal. The two additional vectors are often referred to as *tangent* and *binormal*, although both are actually tangents to the surface.

Once we have the unit vectors $\hat{\mathbf{s}}$ and $\hat{\mathbf{t}}$, the new, bumped surface normal can be computed by modifying the original surface normal $\hat{\mathbf{N}}_0$ according to:

$$\mathbf{N} = \hat{\mathbf{N}}_0 - \frac{\partial h}{\partial s}\hat{\mathbf{s}} - \frac{\partial h}{\partial t}\hat{\mathbf{t}} \tag{2}$$

and renormalizing $\mathbf{N}$ to unit length.

As noted already by Blinn, this is an approximation which is valid only if the local curvature of the surface is small in comparison to the local curvature of the bump function. However, even in situations when the approximation is bad, the visual result remains compelling.

The rather intricate business of fiddling with a 2-D tangent space and its mapping to 3-D space has been the source of much confusion and frustration in the computer graphics industry. Tutorials and examples have been unclear, sometimes even wrong, and even many commercial tools have failed to perform the corresponding transformations and interpolations correctly. For a surprisingly recent and long needed treatise on this, please refer to [2]. The same author later published an article [3] which is essentially a longer and much better motivated version of the presentation that follows.

In short, a procedural bump mapping function defined in 3-D space removes the requirement for an explicit 2-D tangent space, and things become a whole lot simpler.

New style (easy)

Displacement mapping and bump mapping both work the same, except that for bump mapping you only recompute the normals *as if* you displaced the surface. Assume that we use a 3-D function $h(x, y, z)$ for the displacement:

$$\mathbf{P} = \mathbf{P}_0 + h(x, y, z)\hat{\mathbf{N}}_0 \tag{3}$$

The difference from Equation 1 is that there is no longer an explicit 2-D tangent space, just a 3-D object space, and no surface parametrization is implied. To compute the normal of the displaced surface $\mathbf{P}$, we still need to project the gradient of the displacement function to the tangent plane, but this is easily done by basic linear algebra: project the gradient to the normal by a scalar product, and then subtract the part of the gradient which is parallel to the normal. (This is a central part of the *Gram-Schmidt orthogonalization process*, where you construct an orthogonal coordinate basis from an arbitrary set of linearly independent vectors.) What remains after the subtraction is the part of the gradient which is *orthogonal* to the normal, or precisely the gradient projected to the local tangent plane of the surface. However, that vector is now conveniently expressed in 3-D *object space*. It can be added directly to the normal, which is also defined in object space:

$$\mathbf{g} = \nabla h = (\frac{\partial h}{\partial x}, \frac{\partial h}{\partial y}, \frac{\partial h}{\partial z}) \tag{4}$$

$$\mathbf{g}_{\|} = (\mathbf{g} \cdot \hat{\mathbf{N}}_0)\hat{\mathbf{N}}_0 \tag{5}$$

$$\mathbf{g}_{\perp} = \mathbf{g} - \mathbf{g}_{\|} \tag{6}$$

$$\mathbf{N} = \hat{\mathbf{N}}_0 - \mathbf{g}_{\perp} \tag{7}$$

$$\hat{\mathbf{N}} = \frac{\mathbf{N}}{|\mathbf{N}|} \tag{8}$$

Note how this only requires vector computations in object space (x, y, z).

For procedural displacement functions, the gradient $\mathbf{g}$ can often be computed analytically and evaluated exactly with relative ease. If that is not an option, finite differences can always do the trick with sufficient accuracy, albeit mostly at a higher computational cost. Computing $\mathbf{g}$ by finite differences requires at least three extra evaluations of $h(x, y, z)$, whereas the exact analytic gradient of, say, a noise function, often turns out to be considerably less expensive to compute. Simplex noise and Worley noise both clearly fall into this category. Classic Perlin noise is slightly more cumbersome to differentiate analytically, but it's still easier than computing finite differences.

GLSL implementation

In a pair of vertex and fragment shaders in GLSL, assuming that you have a noise function `float noise(vec3 p, out vec3 g)` that returns its gradient in the argument g, displacement and bump mapping could be handled like shown below. Note that the transformation of normals is done by the fragment shader, to make it as straightforward as possible to perform both the displacement and bump computations in object space.

Vertex shader

```
in vec3 position; // Syntax for attributes differs between versions
in vec3 normal;
uniform mat4 modelViewMatrix;
uniform mat4 projectionMatrix;
out vec3 N;         // (For GLSL 1.20, 'out' is 'varying')
out vec3 texcoord; // Object coordinates before displacement

void main()
{
  const float displaceamount = 0.1; // Move max 0.1 up or down
  texcoord = position; // Undisplaced, untransformed position
  // Displace surface
  vec3 p; // Displaced surface point
  vec3 g; // To store gradient of noise
  float d = noise(texcoord*2.0, g);
  g *= 2.0; // Scale gradient with inner derivative
  p = position + displaceamount * d * normal; // Move vertex
  vec3 N_ = g - dot(g, normal) * normal; // Project to tangent plane
  N = normalize(normal - displaceamount * N_); // Perturb normal
    gl_Position = projectionMatrix*modelViewMatrix*vec4(p, 1.0);
}
```

Fragment shader

```
uniform mat3 normalMatrix; // Or fake and use mat3(modelViewMatrix)
in vec3 texcoord;          // (For GLSL 1.20, 'in' is 'varying')
in vec3 N;

void main()
{
  const float bumpamount = 0.05; // Bump 1/2 of disp at scale
  vec3 g; // To store gradient of noise
  float b = noise(texcoord*10.0, g); // Object coordinates
  g *= 10.0; // Scale gradient with inner derivative
  b *= 0.2; g *= 0.2; // Bump noise is 1/5 size of disp noise
  N = normalize(N); // Optional: renormalize after interpolation
  vec3 N_ = g - dot(g, N) * N; // Project to tangent plane
  vec3 bN = N - bumpamount * N_; // Perturb normal
  bN = normalize(normalMatrix * bN); // Transform to view space

  // Perform lighting computations with bN instead of N
  // (not shown here) to generate the fragment shader output
}
```

References

[1] Jim Blinn: *Simulation of wrinkled surfaces*, In ACM Computer Graphics (SIGGRAPH '78), pages 286-292, 1978.

[2] Morten S. Mikkelsen: *Simulation of Wrinkled Surfaces Revisited*, Thesis, 2008.

[3] Morten S. Mikkelsen: *Bump Mapping Unparameterized Surfaces on the GPU*, Journal of Graphics, GPU, and Game Tools, Volume 15, Number 1, pp 49-61, 2010.